Mobile processes & types, part I

Davide Sangiorgi

University of Bologna

Email: Davide.Sangiorgi@cs.unibo.it
http://www.cs.unibo.it/~sangio/

June 25, 2003

Part I: The π-calculus, a calculus of mobile processes

What is the π-calculus

- A **calculus of processes** whose linkage structure may change (mobile processes)

- rich and well-developed theory
- small, yet a high expressive power
- useful type systems
- a metalanguage for concurrent languages (OO, higher-order,...)
- basis for new programming languages

... but also

- A basic **model of computation**, where computing is **interaction**.
  cf. – λ-calculus (functions)
  – Turing Machines (read/write on a store)

Most important ancestors

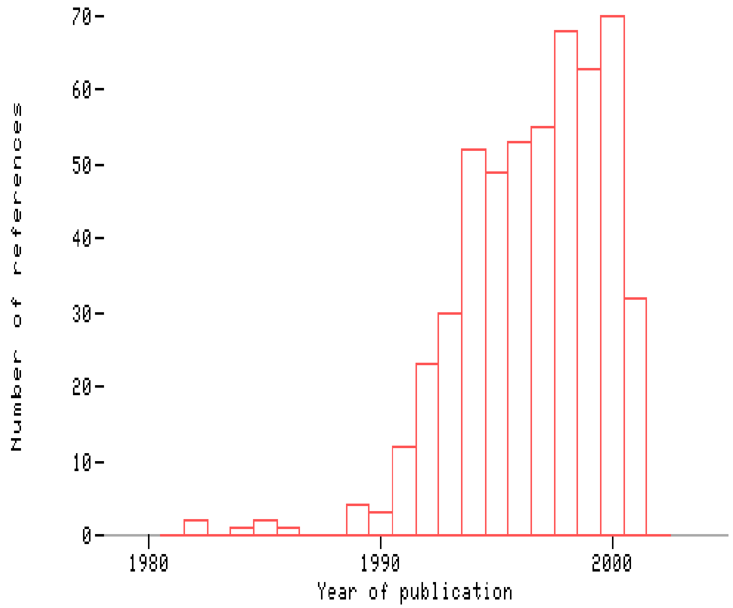
- the λ-calculus (Church, '32-33, '41);
- CCS (Milner, 1980);
- Actors (Hewitt, '77, Agha, '86, '90);
- CCS/PC (Astesiano and Zucca, 1984);
- ECCS (Engberg and Nielsen, 1986).

First π-calculus paper: 1989 (Milner, Parrow, Walker).

“The polyadic π-calculus tutorial” [very influential paper]: 1991 (Milner)

A lot of work in the last 10 years.

Bibliographic statistics (july 2001):

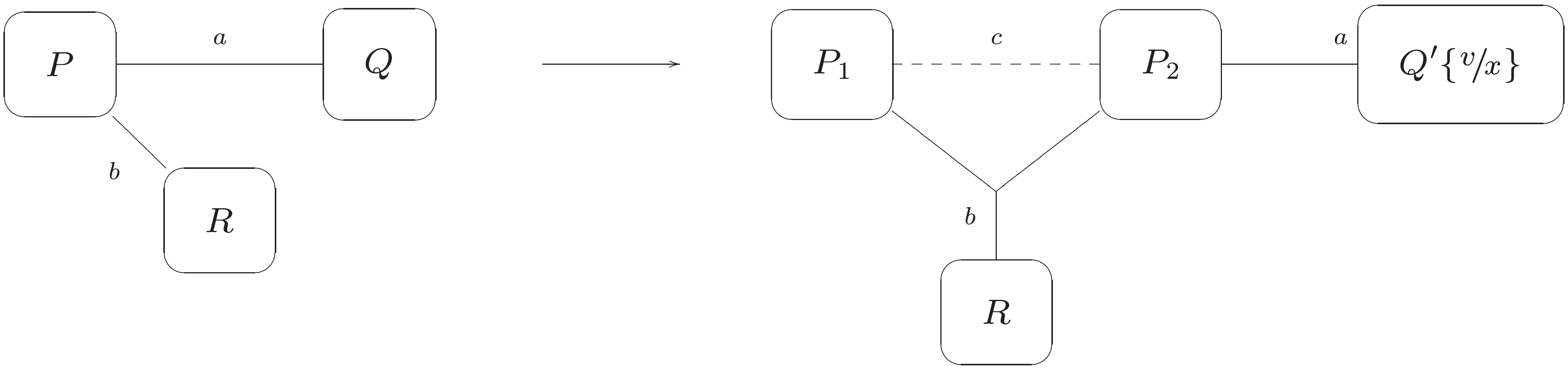


(From `http://lampwww.epfl.ch/mobility/`)

CONTENTS

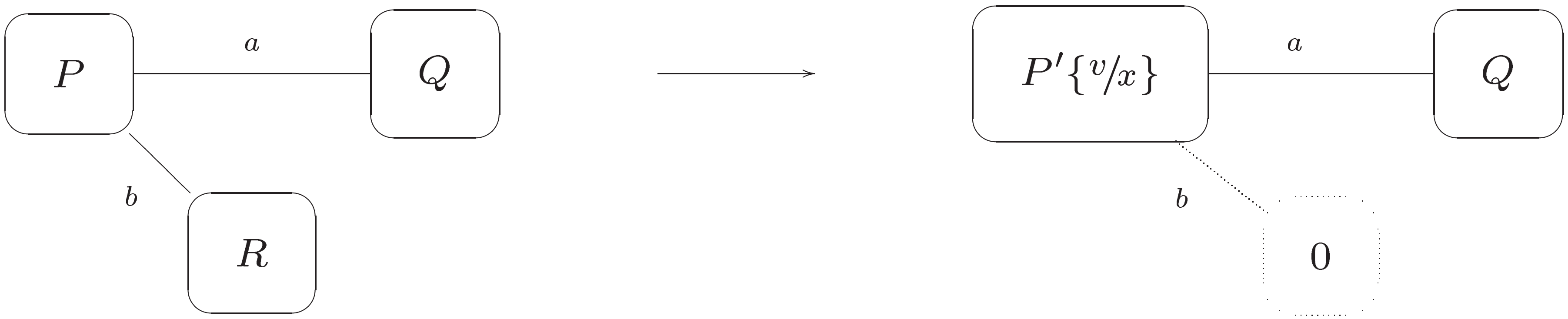
☞ • Mobility in the π-calculus [5]
• The pure (untyped) π-calculus [13]
• Operational semantics [14]
• Programming in the π-calculus [26]
• Types [31]
• The simply-typed π-calculus [33]
• Subtyping [41]
• Types for reasoning [45]
• Advanced type systems [49]
• Other topics [62]

Mobility I: process creation



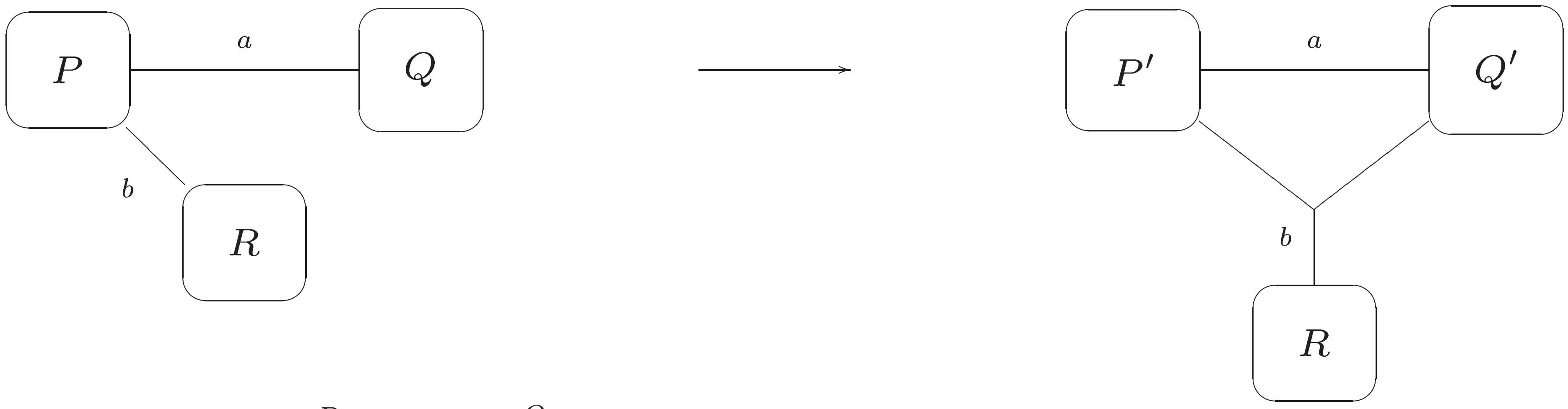
$$\overbrace{\overline{a}\, v \,.\, \boldsymbol{\nu} c\, (P_1 \mid P_2)}^{P} \;\Big|\; \overbrace{a(x).\, Q'}^{Q} \;\Big|\; R \quad \longrightarrow \quad \boldsymbol{\nu} c\, (P_1 \mid P_2) \mid Q'\{{}^{v}\!/\!{}_{x}\} \mid R$$

Mobility I: process termination



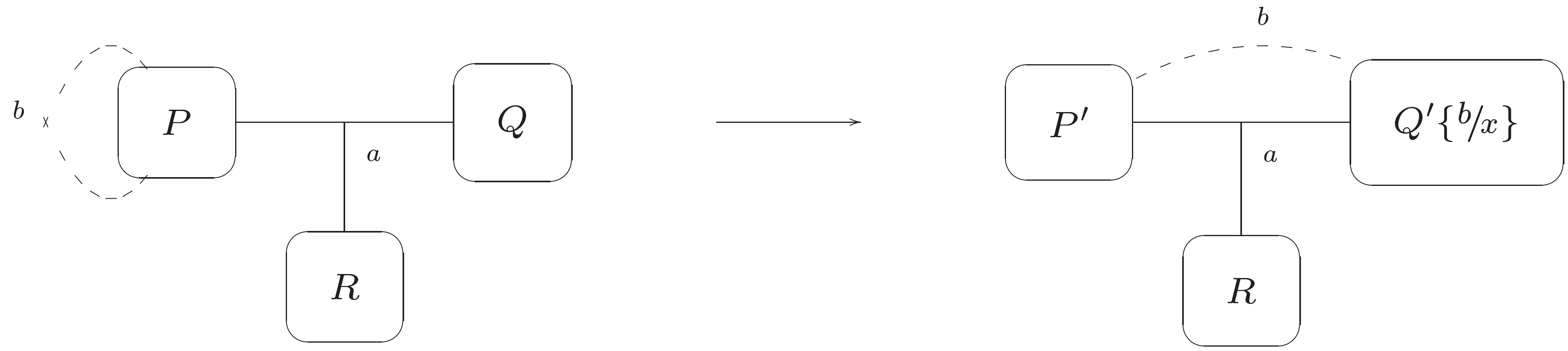
$$\overbrace{b(x).\,P'}^{P} \mid Q \mid \overbrace{\overline{b}\,v\,.\,0}^{R} \longrightarrow P'\{^{v}/_{x}\} \mid Q \mid 0$$

Mobility III: movement of communication links



$$\overbrace{\overline{a}\, b\,.\, P'}^{P} \mid R \mid \overbrace{a(x).\, Q'}^{Q} \longrightarrow P' \mid R \mid Q\{^{b}\!/\!_{x}\}$$

Mobility IV: movement of *private* links



$$\overbrace{(\boldsymbol{\nu} b)\,(\overline{a}\,b\,.\,P')}^{P} \;\Big|\; \overbrace{a(x)\,.\,Q'}^{Q} \;\Big|\; R \qquad \longrightarrow \qquad (\boldsymbol{\nu} b)\,(P' \mid Q'\{^{b}\!/\!_{x}\}) \mid R$$

Mobility in the π-calculus:

- **Name mobility**: Processes communicate via names (channels/links) and names may move (cf: hypertext links, mobile phones, object references, ...)

Another forms of mobility:

- **Process mobility**: Processes communicate via names and processes may move (cf: mobile code, Jini, higher-order languages, load balancing ...)

Furthermore, in the π-calculus there is no explicit notion of location and distribution

The choice of pure name mobility for the π-calculus:

- mathematically simpler
- other forms of mobility may (sometimes) be reduced to it
- basis for more complex formalisms

Reducing process mobility to name mobility (example)

a “higher-order” process

$$\overline{a}\,R\,.\,0 \mid a(x).\,x$$

$$\downarrow$$

$$R$$

... compiled into the π-calculus

$$(\boldsymbol{\nu} q)\ (\overline{a}\,q\,.\,q.\,R) \mid a(x).\,\overline{x}.\,0$$

$$\downarrow$$

$$(\boldsymbol{\nu} q)\ (q.\,R \mid \overline{q}.\,0)$$

$$\downarrow$$

$$(\boldsymbol{\nu} q)\ R \approx R$$

where:

- $a.\,P$ and $\overline{a}.\,P$ are abbreviations for inputs and outputs in which the value communicated is not important
- $\approx$ indicates the application of a simple garbage collection law

Reducing process mobility to name mobility (example 2)

$$\overline{a}\, R \,.\, 0 \mid a(x).\,(x \mid x)$$

$$\downarrow$$

$$R \mid R$$

$$(\boldsymbol{\nu} q)\ (\overline{a}\, q \,.\, !q.\, R) \mid a(x).\,(\overline{x}.\, 0 \mid \overline{x}.\, 0)$$

$$\downarrow$$

$$(\boldsymbol{\nu} q)\ (!q.\, R \mid \overline{q}.\, 0 \mid \overline{q}.\, 0)$$

$$\downarrow$$

$$\downarrow$$

$$(\boldsymbol{\nu} q)\ (R \mid R \mid !q.\, R) \approx R \mid R$$

Correctness of the compilation

Write $\approx$ for behavioural equivalence

Theorem [operational correspondence on reductions] Let P be a “higher-order” process.

1. If $P \longrightarrow P'$, then $[\![P]\!] \longrightarrow\approx [\![P']\!]$;
2. the converse, i.e., if $[\![P]\!] \longrightarrow P''$, then there is P' s.t. $P \longrightarrow P'$ and $P'' \approx [\![P']\!]$.

Theorem [full abstraction] For all “higher-order” processes P, Q it holds that

$$P \approx Q \text{ iff } [\![P]\!] \approx [\![Q]\!].$$

CONTENTS

✓ • Mobility in the π-calculus [5]

☞ • The pure (untyped) π-calculus [13]

• Operational semantics [14]

• Programming in the π-calculus [26]

• Types [31]

• The simply-typed π-calculus [33]

• Subtyping [41]

• Types for reasoning [45]

• Advanced type systems [49]

• Other topics [62]

The π-calculus (core)

$a, b, \ldots, x, y, z, \ldots$			*Names*
			Processes
P	$::=$	0	nil process
	$\mid$	$P \mid P$	parallel
	$\mid$	$(\boldsymbol{\nu} a)\ P$	restriction
	$\mid$	$a(x).\ P$	input
	$\mid$	$\overline{a}\, b\ .\ P$	output
	$\mid$	$!P$	replication ($= P \mid P \ldots$)

- For simplicity (and because less useful), operators of sum and matching are omitted
- Replication replaces recursion

An attempt at labelled transitions (à la CCS)

$\mathtt{inp}$: $\quad a(b).\,P \xrightarrow{a(b)} P \qquad\qquad$ $\mathtt{out}$: $\quad \overline{a}\,b\,.\,P \xrightarrow{\overline{a}\,b} P$

$\mathtt{par}$: $\quad \dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad\qquad$ $\mathtt{com}$: $\quad \dfrac{P \xrightarrow{a(x)} P' \qquad Q \xrightarrow{\overline{a}\,b} Q'}{P \mid Q \xrightarrow{\tau} P'\{b/x\} \mid Q'}$

$\mathtt{res}$: $\quad \dfrac{P \xrightarrow{\alpha} P'}{\boldsymbol{\nu} a\, P \xrightarrow{\alpha} \boldsymbol{\nu} a\, P'} \qquad$ (α is not an action at a)

But then:

- $a(x).\,P \mid x(y).\,Q \xrightarrow{a(x)} P \mid x(y).\,Q$, hence

$$(a(x).\,P \mid x(y).\,Q) \mid \overline{a}\,c\,.\,R \xrightarrow{\tau} (P \mid x(y).\,Q)\{c/x\} \mid R$$

- $\boldsymbol{\nu} b\,\overline{a}\,b\,.\,P \xrightarrow{\overline{a}\,b} \boldsymbol{\nu} b\,P$, hence

$$(\boldsymbol{\nu} b\,\overline{a}\,b\,.\,\overline{b}\,c\,) \mid a(x).\,x(y).\,Q \xrightarrow{\tau} (\boldsymbol{\nu} b\,\overline{b}\,c\,) \mid b(y).\,Q$$

(**dynamic binding!!**)

Labeled transition semantics (amended)

inp: $a(b).\, P \xrightarrow{a(b)} P$

out: $\overline{a}\, b \,.\, P \xrightarrow{\overline{a}\, b} P$

par: $$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \text{ if } \mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$$

com: $$\frac{P \xrightarrow{a(x)} P' \qquad Q \xrightarrow{\overline{a}\, b} Q'}{P \mid Q \xrightarrow{\tau} P'\{^b\!/\!_x\} \mid Q'}$$

close: $$\frac{P \xrightarrow{a(x)} P' \qquad Q \xrightarrow{\overline{a}(b)} Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\nu} b\, (P'\{^b\!/\!_x\} \mid Q')} \text{ if } b \notin \mathrm{fn}(P)$$

res: $$\frac{P \xrightarrow{\alpha} P'}{\boldsymbol{\nu} a\; P \xrightarrow{\alpha} \boldsymbol{\nu} a\; P'}\; a \notin \mathrm{n}(\alpha)$$

open: $$\frac{P \xrightarrow{\overline{a}\, b} P'}{\boldsymbol{\nu} b\; P \xrightarrow{\overline{a}(b)} P'}\; a \neq b$$

rep: $$\frac{P \mid\, !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

alpha: $$\frac{P =_\alpha Q \qquad Q \xrightarrow{\alpha} Q'}{P \xrightarrow{\alpha} Q'}$$

Reduction semantics

[cf: Chemical Abstract Machine, Berry & Boudol '90, Milner '91]

Structural congruence ($\equiv$). The smallest congruence s.t.:

1. $P \equiv Q$ if P and Q are alpha convertible
2. $P \mid 0 \equiv P$, $\quad P \mid Q \equiv Q \mid P$, $\quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
3. $\boldsymbol{\nu} 0 \ \equiv 0$, $\quad \boldsymbol{\nu} x\, \boldsymbol{\nu} y\, P \equiv \boldsymbol{\nu} y\, \boldsymbol{\nu} x\, P$
4. $\boldsymbol{\nu} x\, (P \mid Q) \equiv (\boldsymbol{\nu} P\ ) \mid Q$, if x not free in Q
5. $!P \equiv P \mid\, !P$

Reduction relation ($\longrightarrow$)

$$\text{R-INTER}\quad \frac{}{(\overline{x}\, y\,.\, P_1) \mid (x(z).\, P_2) \longrightarrow P_1 \mid P_2\{y/z\}}$$

$$\text{R-PAR}\quad \frac{P_1 \longrightarrow P_1'}{P_1 \mid P_2 \longrightarrow P_1' \mid P_2} \qquad\qquad \text{R-RES}\quad \frac{P \longrightarrow P'}{\boldsymbol{\nu} z\, P \longrightarrow \boldsymbol{\nu} z\, P'}$$

$$\text{R-STRUCT}\quad \frac{P_1 \equiv P_2 \longrightarrow P_2' \equiv P_1'}{P_1 \longrightarrow P_1'}$$

Reduction semantics: examples

$$
\begin{array}{rcl}
x(y).\, P \mid R \mid \overline{x}\, z \,.\, Q & \equiv & \\
\overline{x}\, z \,.\, Q \mid x(y).\, P \mid R & \longrightarrow & \\
Q \mid P\{^z/_x\} \mid R & \equiv & P \\
\{^z/_x\} \mid R \mid Q & &
\end{array}
$$

Let $P \triangleq \boldsymbol{\nu} y\, (\overline{x}\, y \,.\, 0)$.

$$
\begin{array}{rcl}
x(z).\, \overline{a}\, z \,.\, 0 \mid !P & \equiv & \\
x(z).\, \overline{a}\, z \,.\, 0 \mid \boldsymbol{\nu} y\, (\overline{x}\, y \,.\, 0) \mid !P & \equiv & \\
\boldsymbol{\nu} y\, (x(z).\, \overline{a}\, z \,.\, 0 \mid \overline{x}\, y \,.\, 0) \mid !P & \equiv & \\
\boldsymbol{\nu} y\, (\overline{x}\, y \,.\, 0 \mid x(z).\, \overline{a}\, z \,.\, 0) \mid !P & \longrightarrow & \\
\boldsymbol{\nu} y\, (0 \mid \overline{a}\, y \,.\, 0) \mid !P & \equiv & \\
\boldsymbol{\nu} y\, (\overline{a}\, y \,.\, 0) \mid !P & &
\end{array}
$$

Exercise: contrast the derivations of the interaction at a of the process

$$a(x).\, P \mid \boldsymbol{\nu} b\, (\overline{a}\, b\, .\, Q)$$

in the reduction and in the labelled semantics

Reduction semantics is easier to grasp.

However, labeled semantics:

- says how a process can interact with its environment
- makes it easier to prove properties of behaviour

Lemma If $P \longrightarrow Q$ then there is a derivation of the reduction in which R-STRUCT is applied, if at all, only as the last rule.

Lemma If $P \longrightarrow Q$ then there are x, y, z, $\widetilde{z}$, R_1, R_2 and S such that

$$\begin{array}{rcl} P & \equiv & (\boldsymbol{\nu}\widetilde{z})\,(\overline{x}\,y\,.\,R_1 \mid x(z).\,R_2 \mid S) \\ Q & \equiv & (\boldsymbol{\nu}\widetilde{z})\,(R_1 \mid R_2\{^{y}\!/\!_{z}\} \mid S) \end{array}$$

Theorem $P \longrightarrow P'$ iff $P \xrightarrow{\tau}\equiv P'$.

Behavioural equivalence

An extensional definition of behavioural equivalence:

$P \Downarrow_a \ \triangleq\ P$ is capable of performing an input or output at a.

barbed congruence, $\approx$

Let $\dot{\simeq}$ (**barbed bisimulation**) be the largest symmetric relation on processes such that $P \ \dot{\simeq}\ Q$ implies:

1. if $P \Longrightarrow P'$ then there is Q' such that $Q \Longrightarrow Q'$ and $P' \ \dot{\simeq}\ Q'$;
2. for each channel a, if $P \Downarrow_a$ then also $Q \Downarrow_a$.

Barbed congruence $\triangleq$ the congruence induced by barbed bisimulation.

Problem (for proofs): The universal quantification on contexts.

Possible remedy techniques:

1. Context lemmas
2. *Intensional* characterisations
3. Algebraic laws

Labeled bisimilarity, in CCS and π-calculus

observation equivalence (or weak bisimulation), $\simeq$

$P \simeq Q$ implies

- if $P \stackrel{\mu}{\Longrightarrow} P'$ then there exists Q' s.t. $Q \stackrel{\mu}{\Longrightarrow} Q'$ and $P' \simeq Q'$ (for $\mu \neq \tau$);
- $\ldots$

$\Rightarrow$ equality between the synchronisation trees of P and Q
(precisely: equality as non-well-founded sets).

Theorem In CCS (with guarded sums), barbed congruence and observation equivalence coincide.

Similar characterisations for the untyped and simply-typed π-calculi (**early bisimulation**).

Problem:
beyond simply-typed π-calculus the standard intensional definitions are too strong (research area)

Observation equivalence

- coinductive definition
- no quantification on contexts
- “up to techniques” can enhance the coinduction proof method
- used in tools for mechanically checking equalities
- various characterisations: **non-well founded sets** **[Aczel]**, **modal logic** **[Hennessy, Milner]**, **final coalgebras** **[Rutten, Turi, Honsell, Lenisa, Montanari, Pistore...]** **domain theory** **[Abramsky]**, **open maps** **[Joyal, Nielsen, Winskel, Fiore, Cattani, Power,...]**,

The polyadic π-calculus (first try...)

$a, b, \ldots, x, y, z, \ldots$ *Names*

Processes

P	$::=$	0	nil process
	$\mid$	$P \mid P$	parallel
	$\mid$	$(\boldsymbol{\nu} a)\ P$	restriction
	$\mid$	$a(\widetilde{x}).\,P$	input
	$\mid$	$\overline{a}\langle\widetilde{b}\rangle.\,P$	output
	$\mid$	$!P$	replication

Abbreviations:

$$\overline{x}.\,P \triangleq \overline{x}\langle\rangle.\,P$$

$$x.\,P \triangleq x().\,P$$

$$\overline{x}\langle\widetilde{y}\rangle \triangleq \overline{x}\langle\widetilde{y}\rangle.\,0$$

$$\Longrightarrow \triangleq \longrightarrow^{*}$$

CONTENTS

✓ • Mobility in the π-calculus [5]
✓ • The pure (untyped) π-calculus [13]
✓ • Operational semantics [14]
☞ • Programming in the π-calculus [26]
• Types [31]
• The simply-typed π-calculus [33]
• Subtyping [41]
• Types for reasoning [45]
• Advanced type systems [49]
• Other topics [62]

Example: encoding boolean values

$$
\begin{array}{rcl}
\mathtt{true}_a & \triangleq & a(x,y).\,\overline{x} \\
\mathtt{false}_a & \triangleq & a(x,y).\,\overline{y}
\end{array}
$$

An if-then-else process, meeting $\mathtt{true}_a$ (assume p, q fresh):

$$
\begin{array}{rl}
& \mathtt{true}_a \mid (\boldsymbol{\nu} p, q)\; \overline{a}\langle p, q\rangle.\,(p.\,P \mid q.\,Q) \\
= & a(x,y).\,\overline{x} \mid (\boldsymbol{\nu} p, q)\; \overline{a}\langle p, q\rangle.\,(p.\,P \mid q.\,Q) \\
\longrightarrow & (\boldsymbol{\nu} p, q)\; (\overline{p} \mid p.\,P \mid q.\,Q) \\
\longrightarrow & (\boldsymbol{\nu} p, q)\; (0 \mid P \mid q.\,Q) \\
\approx & P
\end{array}
$$

($\approx$ indicates the application of some simple garbage-collection laws)

Mimicking polyadic communications in the pure π-calculus

Consider a polyadic communication $\quad a(x_1, x_2).\, P \mid \overline{a}\langle b_1, b_2\rangle.\, Q \Longrightarrow P\{\tilde{b}/\tilde{x}\} \mid Q$

Can the above behaviour be mimicked in the monadic π-calculus?

First attempt: $\quad a(x_1).\, a(x_2).\, P \mid \overline{a}\, b_1 \,.\, \overline{a}\, b_2 \,.\, Q \Longrightarrow P\{\tilde{b}/\tilde{x}\} \mid Q$

NO! A third process could interfere using a.

Second attempt: $\quad a(w).\, w(x_1).\, w(x_2).\, P \mid \boldsymbol{\nu} w\, \overline{a}\, w \,.\, \overline{w}\, b_1 \,.\, \overline{w}\, b_2 \,.\, Q \Longrightarrow P\{\tilde{b}/\tilde{x}\} \mid Q$

NB: This encoding of polyadic π-calculus into monadic π-calculus is not fully abstract.

The π-calculus has an amazing expressive power. We can encode in it:

- higher-order communications
- data values
- functions
- object
- ....

It is very convenient in programming to take some simple forms of data as primitives: integers, booleans, ...

(though a basic calculus should keep taxonomy as litte as possible)

Example: the factorial function

$$
\begin{array}{llll}
\mathrm{Fact} & \triangleq & !f(x,r).\ \ \mathtt{if}\ \ x=0 & \mathtt{then}\ \ \overline{r}\langle 1\rangle \\
 & & & \mathtt{else}\ \ (\boldsymbol{\nu} r')\ \overline{f}\langle x-1, r'\rangle .\, r'(m).\, \overline{r}\langle x * m\rangle
\end{array}
$$

$$
\mathrm{Fact} \mid \boldsymbol{\nu} r\ (\overline{f}\langle 3, r\rangle) \Longrightarrow \mathrm{Fact} \mid \overline{r}\langle 6\rangle
$$

Note: FACT can compute answers to several requests in parallel.

Main things seen so far

- mobility
- reduction semantics
- expressiveness of π-calculus, and some simple programming

CONTENTS

✓ • Mobility in the π-calculus [5]
✓ • The pure (untyped) π-calculus [13]
✓ • Operational semantics [14]
✓ • Programming in the π-calculus [26]
☞ • Types [31]
• The simply-typed π-calculus [33]
• Subtyping [41]
• Types for reasoning [45]
• Advanced type systems [49]
• Other topics [62]

Types in Programming languages

- to detect simple kinds of programming errors at compilation time;
- to perform optimisations in compilers;
- to aid the structure and design of systems;
- to compile modules separately;
- to reason about programs;
- etc...

- In **sequential languages**: type systems are widely used and generally well-understood.

- In **concurrent languages**: the tradition of type systems is much less established.

Run-time errors (polyadicity)

$$\underline{\overline{x}\langle y, z\rangle}.\, P \mid \underline{x(h)}.\, Q \longrightarrow \mathtt{wrong}$$

Assigning an arity to each channel is not enough:

$$\begin{array}{rcl} \overline{x}\langle y\rangle.\,\overline{y}\langle z_1, z_2\rangle.\, P \mid x(h).\, h(k).\, Q & \longrightarrow & \\ \underline{\overline{y}\langle z_1, z_2\rangle}.\, P \mid \underline{y(k)}.\, (Q\{^y\!/\!_k\}) & \longrightarrow & \mathtt{wrong} \end{array}$$

Against errors, we need to **type** names.

CONTENTS

- ✓ • Mobility in the π-calculus [5]
- ✓ • The pure (untyped) π-calculus [13]
- ✓ • Operational semantics [14]
- ✓ • Programming in the π-calculus [26]
- ✓ • Types [31]
- ☞ • The simply-typed π-calculus [33]
- • Subtyping [41]
- • Types for reasoning [45]
- • Advanced type systems [49]
- • Other topics [62]

Typed (polyadic) CCS

$a, b, \ldots$			*channels*
$x, y, z \ldots$			*variables*
			Processes
P	$::=$	$a(x).\, P$	input
	$\mid$	$\overline{a}\, v \,.\, P$	output
	$\mid$		tuple destructor
		with $(x_1 \ldots x_n) = v$ do P	
	$\mid$	$\ldots$	$\ldots$

			Values
v	$::=$	x	variables
	$\mid$	$n \mid \ldots$	integers, ...
	$\mid$	$\langle v_1 \ldots v_n \rangle$	tuple of values
	$\mid$	$\ldots$	...
			Types
T	$::=$	$ChT \mid V$	channel, value types
ChT	$::=$	$\sharp\, V$	connection type
V	$::=$	$\mathtt{Int} \mid \ldots$	integers, ...
	$\mid$	$\langle V_1 \ldots V_n \rangle$	tuple types
	$\mid$	$\ldots$	

Type environments

$$\Gamma \quad ::= \quad \Gamma, x : V \mid \Gamma, a : ChT \mid \emptyset$$

Additional operational rules:

$$\mathsf{with}\ (x_1 \ldots x_n) = \langle v_1 \ldots v_n \rangle\ \mathsf{do}\ P \longrightarrow P\{{}^{v_1 \ldots v_n}\!/\!_{x_1 \ldots x_n}\}$$

We can then define as an **abbreviation:**

$$y(x_1 \ldots x_n).\, P \ \triangleq\ y(z).\ \mathsf{with}\ (x_1 \ldots x_n) = z\ \mathsf{do}\ P$$

Typed (polyadic) π-calculus

$a, b, \ldots x, y, z$ — *Names*

Processes

$$
\begin{array}{lcll}
P & ::= & v(x).\,P & \text{input} \\
 & \mid & \overline{v}\,v\,.\,P & \text{output} \\
 & \mid & & \text{tuple destructor} \\
 & & \text{with } (x_1..\,x_n) = v \text{ do } P & \\
 & \mid & \ldots & \ldots
\end{array}
$$

Values

$$
\begin{array}{lcll}
v & ::= & a \mid n \mid \ldots & \text{names, integers, ...} \\
 & \mid & \langle v_1..\,v_n \rangle & \text{tuple of values} \\
 & \mid & \ldots & \text{...}
\end{array}
$$

Types

$$
\begin{array}{lcll}
T & ::= & \sharp\, T & \text{channel type} \\
 & \mid & \mathtt{Int} \mid \ldots & \text{integers, ...} \\
 & \mid & \langle T_1..\,T_n \rangle & \text{tuple type} \\
 & \mid & \ldots &
\end{array}
$$

Type environments

$$
\Gamma \quad ::= \quad \Gamma, a : T \mid \emptyset
$$

Type judgements

$$\Gamma \vdash v : T \qquad\qquad\qquad \Gamma \vdash P$$

Meaning:

"value v has type T under the type assignment for names Γ",

"process P respects the type assignment for names Γ".

Notation: In this talk, we are not concerned about type inference,
$\Rightarrow$ sometimes add type annotations to bound names.

A sample of typing rules

values:

$$\frac{\Gamma(v) = T}{\Gamma \vdash v \ : T} \qquad \frac{\Gamma \vdash v_i \ : T_i}{\Gamma \vdash \langle v_1 . . v_n \rangle \ : \langle T_1 . . T_n \rangle}$$

processes:

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \frac{\Gamma \vdash v \ : \langle T_1 . . T_n \rangle \qquad \Gamma,\ x_1 : T_1 . . x_n : T_n \vdash P}{\Gamma \vdash \ \mathsf{with}\ (x_1 . . x_n) = v\ \mathsf{do}\ P}$$

$$\frac{\Gamma \vdash w \ : \sharp T \qquad \Gamma \vdash v \ : T \qquad \Gamma \vdash P}{\Gamma \vdash \overline{w}\, v\,.\, P} \qquad \frac{\Gamma \vdash w \ : \sharp T \qquad \Gamma, y : T \vdash P}{\Gamma \vdash w(y).\, P}$$

π-calculus is hardly ever used untyped!

Types can be enriched in many ways.

- **Familiar type constructs** (in place of, or in addition to, products), ex: union, records, variants, basic values, functions, linearity, polymorphism, etc.
- Type constructs **specific to processes**, ex: process-passing, receptiveness, deadlock-freedom, sessions, etc.
- **Types are assigned to names**

Example: adding union types

Additional productions in the grammars

			Values
v	$::=$	$\mathsf{inl\_}v$	union value, left
	$\mid$	$\mathsf{inr\_}v$	union value, right
			Processes
P	$::=$	$\mathsf{case}\ v\ \mathsf{of}\ [\,\mathsf{inl\_}(x_1)\ \triangleright\ P_1\,;\ \mathsf{inr\_}(x_2)\ \triangleright\ P_2\,]$	union case
			Types for values
V	$::=$	$V_1 \oplus V_2$	union type

Additional typing rules

$$\texttt{T-with}\frac{\Gamma \vdash v \ : T_1 \oplus T_2 \qquad \Gamma, x_i \ : \ T_i \vdash P_i \qquad i = 1,2}{\Gamma \vdash \mathsf{case}\ v\ \mathsf{of}\ [\,\mathsf{inl\_}(x_1) \ \triangleright \ P_1\,;\ \mathsf{inr\_}(x_2) \ \triangleright \ P_2\,]}$$

$$\texttt{Tv-uniL}\frac{\Gamma \vdash v \ : T_1}{\Gamma \vdash \mathsf{inl\_}v \ : T_1 \oplus T_2}$$

$$\texttt{Tv-uniR}\frac{\Gamma \vdash v \ : T_2}{\Gamma \vdash \mathsf{inr\_}v \ : T_1 \oplus T_2}$$

Additional transition rules

$$\texttt{UniL}\frac{}{\mathsf{case}\ \mathsf{inl\_}v\ \mathsf{of}\ [\,\mathsf{inl\_}(x_1) \ \triangleright \ P_1\,;\ \mathsf{inr\_}(x_2) \ \triangleright \ P_2\,] \xrightarrow{\tau} P_1\{{}^{v}\!/\!{}_{x_1}\}}$$

$$\texttt{UniR}\frac{}{\mathsf{case}\ \mathsf{inr\_}v\ \mathsf{of}\ [\,\mathsf{inl\_}(x_1) \ \triangleright \ P_1\,;\ \mathsf{inr\_}(x_2) \ \triangleright \ P_2\,] \xrightarrow{\tau} P_2\{{}^{v}\!/\!{}_{x_2}\}}$$

CONTENTS

✓ • Mobility in the π-calculus [5]
✓ • The pure (untyped) π-calculus [13]
✓ • Operational semantics [14]
✓ • Programming in the π-calculus [26]
✓ • Types [31]
✓ • The simply-typed π-calculus [33]
☞ • Subtyping [41]
• Types for reasoning [45]
• Advanced type systems [49]
• Other topics [62]

Channel types ($\sharp\, T$) are refined by introducing:

$\mathsf{i}\, T$	input (read) capability
$\mathsf{o}\, T$	output (write) capability

$\Rightarrow$ basis for **subtyping**

Example: $x : \mathsf{o}\,\mathsf{o}\, T \vdash (\boldsymbol{\nu} y : \sharp\, T)\,(\overline{x}\, y\, .\, !y(z).\, 0)$ (true, since $\sharp\, T < \mathsf{o}\, T$)

N.B.: The recipient of y cannot **interfere** with the existing input.

$\mathsf{i}\, -$	$\Rightarrow$	covariance
$\mathsf{o}\, -$	$\Rightarrow$	contravariance
$\sharp\, -$	$\Rightarrow$	invariance

Sample of rules for subtyping

Subtyping rules

$$\texttt{Sub-}\sharp\texttt{I}\,\frac{}{\sharp T \leq \mathsf{i}\, T} \qquad \texttt{Sub-}\sharp\texttt{O}\,\frac{}{\sharp T \leq \mathsf{o}\, T}$$

$$\texttt{Sub-II}\,\frac{S \leq T}{\mathsf{i}\, S \leq \mathsf{i}\, T} \qquad \texttt{Sub-OO}\,\frac{T \leq S}{\mathsf{o}\, S \leq \mathsf{o}\, T} \qquad \texttt{Sub-BB}\,\frac{T \leq S \qquad S \leq T}{\sharp S \leq \sharp T}$$

Typing rules

$$\texttt{T-InpS}\,\frac{\Gamma \vdash a : \mathsf{i}\, S \qquad \Gamma, x : S \vdash P}{\Gamma \vdash a(x).\, P}$$

$$\texttt{T-OutS}\,\frac{\Gamma \vdash a : \mathsf{o}\, T \qquad \Gamma \vdash w : T \qquad \Gamma \vdash P}{\Gamma \vdash \overline{a}\, w\,.\, P}$$

$$\texttt{Subsumption}\,\frac{\Gamma \vdash v : S \qquad S \leq T}{\Gamma \vdash v : T}$$

Analogy with mutable variables

$\texttt{ref}\ T$, like $\sharp T$, is invariant.

A variable is used in two ways: $\begin{cases} \text{R-value (read)} \\ \text{L-value (write)} \end{cases}$

If we separate them $\Rightarrow$ the same subtyping as for channels. **[Reynolds's Forsythe]**

Some properties of typing and subtyping

Narrowing:

If $\Gamma, p : S \vdash P$ and $T < S$, then also $\Gamma, p : T \vdash P$

Subject Reduction:

Suppose $\Gamma \vdash P$ and Γ is closed.

- If $P \longrightarrow P'$, then $\Gamma \vdash P'$.

CONTENTS

✓ • Mobility in the π-calculus [5]
✓ • The pure (untyped) π-calculus [13]
✓ • Operational semantics [14]
✓ • Programming in the π-calculus [26]
✓ • Types [31]
✓ • The simply-typed π-calculus [33]
✓ • Subtyping [41]
☞ • Types for reasoning [45]
• Advanced type systems [49]
• Other topics [62]

Types for reasoning

Types: *contracts* between a process and its environment
$\Rightarrow$ reduces the number of 'legal' observer.
$\Rightarrow$ more process equalities.

Behavioural equivalence:

Two processes are *behaviourally equivalent* if no observer can tell them apart.

Notation for behavioural equivalence (a *weak* equivalence):

$$P \approx_\Gamma Q \qquad \begin{cases} \Gamma \vdash P, Q\ ; \\ \text{if the language is untyped, } \Gamma \text{ is trivial} \end{cases}$$

The testing observer must respect the types in Γ

Semantic consequences of i/o types

Example: a printer with mobile ownership

$$\boldsymbol{\nu} p \ \ (P \mid C_1 \mid C_2)$$

where

- P is a printer, accessible via p
- C_1 and C_2 two clients: $\begin{array}{lcl} C_1 & \triangleq & \overline{p}\, j_1 \,.\, \overline{p}\, j_2 \,.\, 0 \\ C_2 & \triangleq & \overline{b}\, p \,.\, 0 \end{array}$

Statement: C_1's print jobs are eventually received and processed in that order by the printer (assume some fairness conditions on execution).

False! In the environment there could be a malicious process such as $\quad C_3 \triangleq b(x).\, x(j).\, 0$

Solution: – the printer has only the input capability on p,
– the initial clients C_1 and C_2 have only the output capability.

Example 2: the factorial function

$$\mathrm{Fact} \triangleq\ !f(x,r).\ \ \mathtt{if}\ x=0\ \ \mathtt{then}\ \overline{r}\,1$$
$$\qquad\qquad\qquad\qquad\qquad\quad \mathtt{else}\ (\boldsymbol{\nu} r')\,(\overline{f}\langle x-1,r'\rangle.\,r'(m).\,\overline{r}\langle x*m\rangle)$$

$$\begin{array}{lll}\mathrm{FactEff} & \triangleq & !f(x,r). \\ & & \ \mathtt{if}\ x=0\ \mathtt{then}\ \overline{r}\,1 \\ & & \ \mathtt{else}\quad \mathtt{if}\ x=1\ \mathtt{then}\ \overline{r}\,1 \\ & & \qquad\quad\ \mathtt{else}\quad \mathtt{if}\ x=2\ \mathtt{then}\ \overline{r}\,2 \\ & & \qquad\qquad\qquad\ \ \mathtt{else}\ \boldsymbol{\nu} r'\,(\overline{f}\langle x-1,r'\rangle \mid r'(m).\,\overline{r}\langle x*m\rangle)\end{array}$$

In a simply-typed π-calculus: $\mathrm{Fact} \not\approx_\Gamma \mathrm{FactEff}$ (for all legal Γ)

With i/o types we can protect the input end of the function:

$\mathrm{SPEC} \approx_{f\leftarrow o} \mathrm{Fact}$ ($\approx_{f\leftarrow o} \triangleq$ the surrounding observer has only the output capability on f)

i/o **subtyping validates the subtyping rule for functions:**

Following Milner,

$$[\![\lambda x.\, M]\!]_p \stackrel{\triangle}{=} (\boldsymbol{\nu} x)\, \overline{p}\, x\, .\, !x(z, y).\, [\![M]\!]_y$$

If $\lambda x.\, M : S \to T$, then $p : \mathsf{o}\, \mathsf{o}\, \langle [\![S]\!], \mathsf{o}\, [\![T]\!] \rangle$

which gives this translation on arrow types:

$$[\![S \to T]\!] \stackrel{\triangle}{=} \mathsf{o}\, \langle [\![S]\!], \mathsf{o}\, [\![T]\!] \rangle$$

where $[\![S]\!]$ is in contravariant position, $[\![T]\!]$ is in covariant position.

CONTENTS

✓ • Mobility in the π-calculus [5]
✓ • The pure (untyped) π-calculus [13]
✓ • Operational semantics [14]
✓ • Programming in the π-calculus [26]
✓ • Types [31]
✓ • The simply-typed π-calculus [33]
✓ • Subtyping [41]
✓ • Types for reasoning [45]
☞ • Advanced type systems [49]
• Other topics [62]

Linearity

We impose *multiplicity* constraints, in addition to the *polarity* constraints of the i/o types

Linear types:

$\ell\mathsf{o}\, T$:	the capability of transmitting a value of type T along that name once.
$\ell\mathsf{i}\, T$:	the capability of receiving a value of type T once.
$\ell\sharp\, T$:	$\ell\mathsf{i}\, T \cup \ell\mathsf{o}\, T$.

Example:

$$\begin{array}{lll} a : \ell\mathsf{o}\, T \,,\, b : T \,,\, c : \sharp\,\ell\mathsf{o}\, T & \vdash & \overline{a}\, b \\ & \vdash & \overline{c}\, a \\ & \vdash & \overline{c}\, a \mid c(x).\, \overline{x}\, b \end{array}$$

Typing rules

A key rule to understand is that for $\Gamma \vdash P_1\ P_2$

$\Rightarrow$ each linear capability in Γ should be exercised exactly once, either in P_1 or in P_2

$\Rightarrow$ the linear capabilities in Γ are split as $\Gamma = \Gamma_1 \uplus \Gamma_2$

$$\frac{\Gamma_1 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 \uplus \Gamma_2 \vdash P_1 \mid P_2}$$

Example

$$\dfrac{\dfrac{\dots}{a : \ell\mathsf{i}\,\mathtt{unit},\ b : \ell\mathsf{o}\,\mathtt{unit} \vdash a(x).\,\overline{b}\,x} \qquad \dfrac{a : \ell\mathsf{o}\,\mathtt{unit} \vdash a : \ell\mathsf{o}\,\mathtt{unit} \qquad \emptyset \vdash \star : \mathtt{unit}}{a : \ell\mathsf{o}\,\mathtt{unit} \vdash \overline{a}\,\star}}{a : \ell\sharp\,\mathtt{unit},\ b : \ell\mathsf{o}\,\mathtt{unit} \vdash a(x).\,\overline{b}\,x \mid \overline{a}\,\star}$$

Combination of types

$$\begin{array}{rcll} \ell\mathsf{i}\, T \uplus \ell\mathsf{o}\, T & \triangleq & \ell\sharp\, T & \\ T \uplus T & \triangleq & T & \text{if } T \text{ is not a linear type} \\ T \uplus S & \triangleq & \texttt{error} & \text{otherwise} \end{array}$$

Combination of type environments

$$(\Gamma_1 \uplus \Gamma_2)(x) \triangleq \begin{cases} \Gamma_1(x) \uplus \Gamma_2(x) & \text{if both } \Gamma_1(x) \text{ and } \Gamma_2(x) \text{ are defined} \\ \Gamma_1(x) & \text{if } \Gamma_1(x)\text{, but not } \Gamma_2(x)\text{, is defined} \\ \Gamma_2(x) & \text{if } \Gamma_2(x)\text{, but not } \Gamma_1(x)\text{, is defined} \\ \text{undefined} & \text{if both } \Gamma_1(x) \text{ and } \Gamma_2(x) \text{ are undefined} \end{cases}$$

Some other rules

$$\frac{\Gamma_1 \vdash v : \mathtt{o}\, S \qquad \Gamma_2 \vdash w : S \qquad \Gamma_3 \vdash P}{\Gamma_1 \uplus \Gamma_2 \uplus \Gamma_3 \vdash \overline{v}\, w \,.\, P}$$

For 0 and !, we check that the type environment has no pending linear capability (not exercised by 0 and over exercised by $!P$)

$$\frac{\Gamma \vdash P}{\Gamma \vdash !P}\ \mathtt{Lin}(\Gamma) = \emptyset \qquad\qquad \frac{}{\Gamma \vdash 0}\ \mathtt{Lin}(\Gamma) = \emptyset$$

A behavioural property of linearity

A τ-insensitiveness properties: certain reductions do not affect behaviour.

Useful for reducing the state space of a process when analysing its behaviour.

Theorem Suppose $\Gamma \vdash P$ and $P \xrightarrow{\tau} P'$ is a communication at an linear name (sum-free). Then $\Gamma \triangleright P \approx P'$.

Channel usage

Besides polarity and multiplicities, we can check the *order* in which channels are used

$(U)T$: a channel that carries values of type T and that is used according to U

U	$::=$	0	not used
	$\mid$	$\mathsf{i}.\,U$	used for input, and then U
	$\mid$	$\mathsf{o}.\,U$	used for output, and then U
	$\mid$	$U_1 \mid U_2$	used as U_1 and U_2, possibly in parallel
	$\mid$	$U_1 \,\&\, U_2$	used as U_1 or U_2
	$\mid$	$*\, U$	used as U arbitrarily many times

A linear type $\ell_\sharp\, T$ is encoded as $(\mathsf{i} \mid \mathsf{o})T$

Example: locks

An important property of locks: when acquired, a lock is eventually released, exactly once.

Type for locks: $(\mathsf{o} \mid *\,\mathsf{i}.\,\mathsf{o})\mathtt{unit}$

Respected here:

$$\overline{lock} \mid lock.\,\langle CS1\rangle.\,\overline{lock} \mid lock.\,\langle CS2\rangle.\,\overline{lock}$$

Not respected here:

$$\overline{lock} \mid lock.\;\; \mathtt{if}\;\; n = 0 \;\;\mathtt{then}\;\; 0 \;\;\mathtt{else}\;\; \overline{a}.\,\overline{lock}$$

$$\overline{lock} \mid lock.\,(\overline{a}.\,\overline{lock} \mid \overline{lock})$$

Uniform receptiveness

A local environment:

$$\boldsymbol{\nu} x \; (!x(y).\, R \mid Q)$$

(Q and R have only the output capability on x)

Name x is “stateless”, and its input end is always available.
$\Rightarrow$ x is *uniformly receptive*.

Uniform receptiveness occurs when modelling functions, objects, communications of processes, RPC protocols, etc...

In π-calculus-based programming languages (Pict, Join, ...): $\mathsf{def}\ x(y) = R\ \mathsf{in}\ Q$

A simple variant: *linear receptiveness*: x is used at most once.

Semantic consequences of receptive types: some laws

if x is uniformly receptive:

$$\overline{y}\,x\,.\,P \quad \approx_\Gamma \quad (\boldsymbol{\nu} z)\,\overline{y}\,z\,.\,(z \succ x \mid P)$$

where z is fresh

$z \succ x \;\triangleq\; !z(a).\,\overline{x}\,a$ *(a link)*

(an output of a global name becomes an output of a fresh name)

Not true in the ordinary π-calculus because of contexts like

$$y(u).\,\overline{u}\,c\,.\,\overline{p} \mid [\cdot]$$

If x is uniformly receptive in Γ:

$$\boxed{\overline{x}\, v \,.\, P \approx_\Gamma \overline{x}\, v \mid P}$$

(makes a synchronous communication into an asynchronous one)

If $P \longrightarrow P'$ by means of a communication at x, then

$$P \approx_\Gamma P'$$

(insensitiveness to internal reductions)

Note: all names uniformly receptive $\Rightarrow$ **confuence**

A distributivity law for local environments:

$$
\begin{array}{rll}
\mathsf{def}\ x(y) = R\ \mathsf{in}\ (P \mid Q) & \approx_{\Gamma} & \mathsf{def}\ x(y) = R\ \mathsf{in}\ P \\
& \mid & \mathsf{def}\ x(y) = R\ \mathsf{in}\ Q
\end{array}
$$

i/o or receptive types are necessary for its validity.

More factorial functions:

$r : \ell\mathtt{rec}(\mathtt{Int}) \triangleq r$ is a linear receptive channel that carries integers

$$
\begin{aligned}
S1 \quad \triangleq \quad & !f(x:\mathtt{Int}, y:\mathtt{Int}, r:\ell\mathtt{rec}(\mathtt{Int})).\\
& \qquad \mathtt{if}\ \ x = 0 \quad \mathtt{then}\ \overline{r}\, y\\
& \qquad \qquad \qquad \quad \mathtt{else}\ \ (\boldsymbol{\nu} r')\, (\overline{f}\langle x-1, x*y, r'\rangle.\, r'(m).\, \overline{r}\, m\,)
\end{aligned}
$$

$$
\begin{aligned}
S2 \quad \triangleq \quad & !f(x:\mathtt{Int}, y:\mathtt{Int}, r:\ell\mathtt{rec}(\mathtt{Int})).\\
& \qquad \mathtt{if}\ \ x = 0 \quad \mathtt{then}\ \overline{r}\, y\\
& \qquad \qquad \qquad \quad \mathtt{else}\ \ \overline{f}\langle x-1, x*y, r\rangle
\end{aligned}
$$

Taking account of the receptiveness of f, r and r':
$\Rightarrow S1$ and $S2$ become behaviourally equivalent. [**Tail-call optimisation**]

Types: some references

Polyadic pi-calculus: Milner 91.

i/o **types**: Pierce and Sangiorgi 93, Boreale and Sangiorgi 98, Merro 98. **Extensions**: De Nicola, Ferrari and Pugliese 98, Sewell 98, Hennessy and Riely 98-9, Gay and Hole 99, Röckl 99, ..

Linearity: Honda 96, Kobayashi, Pierce and Turner 96, Niehren 96, ..

Receptiveness: Pierce and Turner 96, Sangiorgi 97, Kobayashi 97, Amadio, Boudol, Lhoussaine 00

Uage types (including: lock, race, deadlock freedom: Yoshida 96, Boudol 97, Kobayashi, Sumii 97/98/00/02, Abadi, Flanagan, Freund 99/00 (related to the work on π)

Types for guaranteeing secrecy properties: Abadi, Blanchet 97-.., Gordon, Jeffrey 00-..

CONTENTS

✓ • Mobility in the π-calculus [5]

✓ • The pure (untyped) π-calculus [13]

✓ • Operational semantics [14]

✓ • Programming in the π-calculus [26]

✓ • Types [31]

✓ • The simply-typed π-calculus [33]

✓ • Subtyping [41]

✓ • Types for reasoning [45]

✓ • Advanced type systems [49]

☞ • Other topics [62]

Variants of the π-calculus

- asynchronous π-calculus (Aπ): $\overline{a}\, v$ **, no sum**

– easier to implement and closer to distributed system practice
– a "functional" programming style

- locality, Lπ : **Disallow inputs of x in the body P of $a(x).\, P$**

– an "object-oriented" programming style
– algebraic theory (richer than Aπ)
– type inference à la ML

- Join calculus: **Aπ + Lπ + join inputs**

– unique location for channel inputs
– a sharper "functional" programming style
– join inputs

Experimental programming languages

- Pict (Pierce and Turner, '95)
- Facile (projct ECRC, Munich)
- Join (Levy's group at INRIA Rocquencourt)
- Oz (Smolka's group in Saarbrück)
- Piccola (Nierstrasz's group in Berne),
- Klaim (De Nicola's group in Florence)
- TyCO (Vasconcelos, Ravara, et al. in Lisbon)

Verification tools

- Mobility Workbench (Moller, Victor, 94-..)
- HAL (Pisa group, '98-..)
- theorem prover formalisations (Hirschkoff '98, Honsell, Miculan, and Scagnetto '99, Roeckl'00)

Security

Calculi for the specification and verification of security protocols:

- the spi-calculus (Abadi and Gordon, Blanchet '97-..), sjoin (Abadi, Fournet, Gonthier, '99), Applied pi (Abadi, Fournet, ,01)
- probabilistic poly-type process calculi (Lincoln, Mitchell, Shedrov '98-..)

Books

– Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
– Davide Sangiorgi and David Walker. *The pi-calculus: a theory of mobile processes*. Cambridge University Press, 2001.

For information on calculi of mobile processes

`http://lampwww.epfl.ch/mobility/`

A discussion forum:

`http://www-sop.inria.fr/mimosa/personnel/Davide.Sangiorgi/moca.html`